Proceedings of the International Conference on
Formal Methods and Foundations of Artificial Intelligence
Eszterházy Károly Catholic University
Eger, Hungary, June 5–7, 2025
pp. 51–64

DOI: 10.17048/fmfai.2025.51

Beyond Hello World: Teaching software engineering with realistic and automated assignments*

Mihály Dobos-Kovács, András Vörös, Zoltán Micskei

Department of Artificial Intelligence and Systems Engineering
Budapest University of Technology and Economics
Műegyetem rkp. 3., H-1111 Budapest, Hungary
{mdobosko,vori,micskeiz}@mit.bme.hu

Abstract. Software engineering is a complex discipline that requires engineers to blend various skills to produce quality software adeptly. In this paper, we propose a software engineering assignment that follows the lifecycle of a feature of a real-world project, mimics real-world challenges, promotes best practices, and shows the importance of verification techniques. We deploy the assignment in a university course and discuss our findings regarding functional correctness, code quality, and being on schedule. Finally, we propose an AI-assisted outcome estimation method to help identify struggling students while the home assignment is ongoing.

Keywords: software engineering, software engineering education, static analysis, testing techniques, verification techniques, AI in education

AMS Subject Classification: 68N30

1. Introduction

Software engineering is a complex discipline that requires engineers to blend various skills to produce quality software adeptly. These capabilities [21, 23] include proficiency in programming languages and version control, writing high-quality

*The project supported by the Doctoral Excellence Fellowship Programme (DCEP) is funded by the National Research Development and Innovation Fund of the Ministry of Culture and Innovation and the Budapest University of Technology and Economics, under a grant agreement with the National Research, Development and Innovation Office.

code using static analysis within continuous integration frameworks, conducting code reviews, and numerous verification techniques in either a traditional [17] or agile [3] development workflows [4]. However, university assignments are typically oversimplified and do not mirror real-world challenges. Usually, they only focus on a couple of selected aspects, while real-world tasks require engineers to apply the best practices of numerous aspects simultaneously.

Recent research [16] highlights the importance of using problem-based learning, gamification, and automated feedback to teach the basics of software engineering, software quality, and testing. In this paper, we combine these insights with agile methods to build a workflow that follows the lifecycle of a feature of a real-world project, mimics real-world challenges, promotes best practices, and shows the importance of verification techniques at different steps of the workflow, while enabling continuous feedback using automation.

Section 2 describes our proposed assignment, while in Section 3, we evaluate its deployment in a university course. Section 4 introduces an AI-assisted outcome estimation approach that builds on our previous findings. In Section 5, we present the relevant related work in the field of software engineering education. Finally, in Section 6, we conclude our work.

2. Overview of assignment

Figure 1 depicts our proposed assignment workflow. Students are randomly divided into two groups: Variant A or B. Initially, they complete an onboarding task to set up their development environment, including the build and debugging environments, as well as Git configuration. Subsequently, students implement their assigned variant according to a detailed specification, introducing a new feature into a preexisting software. Post-implementation, based on this specification, they create a test suite for the opposite variant. Finally, students are paired randomly for peer code reviews using these tests, ensuring each student provides and receives feedback.

Throughout this simplified development workflow, students engage with various verification techniques. (1) During the implementation phase (Figure 2), they are guided by quality assurance techniques, starting with the writing unit tests

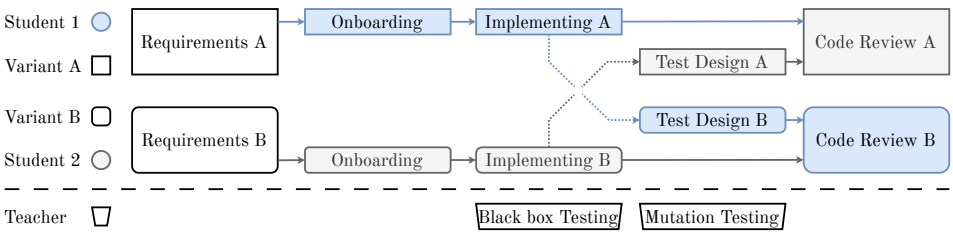


Figure 1. The proposed assignment workflow.

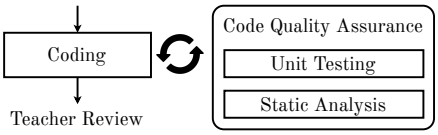


Figure 2. The implementation step in more detail.

guided by coverage metrics. Static analysis tools then provide continuous feedback on coding standards and code smells. (2) In the test design phase, students use specification-based test design methods to design a test suite. (3) Ultimately, students perform code reviews, utilizing insights from quality assurance tools and their test suites.

For educators, student submissions undergo constant automated evaluation. Implementations are verified with a (private) hidden test suite derived from the requirements using specification-based testing methods. Students' test suites are analyzed through mutation testing [8], comparing them against a perfect implementation to gauge correctness and against intentionally flawed implementations to assess completeness.

We implemented the proposed approach in the open-source Feseip[1] framework. Students use Git for version control, GitHub for collaboration and reviews, and SonarQube for quality management. Following GitHub flow, students submit a pull request, request review with a GitHub comment when ready, and obtain automated feedback via a GitHub Check. Architecturally, a separate database maintains necessary information and project state, while teacher code for evaluation never leaves the server of Feseip.

During the assignment, students are tasked to extend OpenMetroMaps[2][3] with a new feature. This open-source 25k+ LOC Java application renders and modifies schematic public transport maps. It also allows importing maps from standardized and well-known formats such as GTFS[4] [10] and OpenStreetMaps data.

The assignment spanned about 7 weeks, with an additional week allowed for late submissions. Students had designated periods: 1 week for onboarding, 2 weeks for implementation, 2 weeks for test design, and 2 weeks for the review phase. Meeting these suggested deadlines earned bonus points. Additional bonus points were granted for achieving adequate code quality, verified by passing SonarQube's default quality gate, and for excellent functional correctness, measured by the performance on our hidden test suite. During these 8 weeks, students received daily feedback.

[1] https://github.com/ftsrg-edu/feseip
[2] https://github.com/ftsrg-softeng/openmetromaps
[3] https://www.openmetromaps.org/
[4] https://gtfs.org/

Listing 1. An excerpt from one of the variants' specifications.

A1 Merge Stops:

1. The user selects exactly two stops. The order of selection matters. The stop first selected is called the *primary stop*, and the stop selected second is the *secondary stop*.
2. As a result of the merge, the *primary stop* is modified, and the *secondary stop* is removed from the model.
3. All lines that have a stop at the *secondary stop* will stop at the *primary stop* after the merge.
4. If a line stops at both the *primary* and *secondary stops*:
   (a) If the two selected stops are adjacent (with no other stop in between), the segment defined by the two adjacent stops is removed from the line.
   (b) If the two selected stops are not adjacent (with at least one stop in between), the merge operation is not executed on any line.
   (c) There must be at least two stops on each line. If this is not ensured as a result of the merge operation, the operation will not be executed.

Example Assignment. Students receive a GitHub issue containing the requirements for their variant, with pre-itemized requirements for better clarity. Listing 1 provides an excerpt from such a specification. Typically, students are expected to write 150-200 lines of code. Listing 1 contains one out of five operations from that variant, requiring 40-60 lines of code. The students' code must adhere to a specific interface to enable automatic evaluation. The primary challenge is not coding per se, but analyzing the 25,000 lines of OpenMetroMaps code and comprehending its underlying data structure, before being able to contribute the 200 lines this new feature requires.

In the test design phase, students apply specification-based techniques to create tests. They utilize the arrange, act, assert structure, where standard GTFS files represent the arrange and assert components detailing a map's state pre- and post-operation, and a text file describes the operation. Utilizing GTFS files enables students to practice with established standards unfamiliar to them.

3. Evaluation

In 2024, the assignment was launched within an undergraduate course, engaging around 700 students in Hungarian, English, and German languages. The task commenced on October 8$^{\text{th}}$, with optional phase deadlines: onboarding by October 15$^{\text{th}}$, implementation by October 29$^{\text{th}}$, test design by November 12$^{\text{th}}$, and review by November 26$^{\text{th}}$. The final submission deadline was November 28$^{\text{th}}$, with late submissions[5] accepted until December 5$^{\text{th}}$.

Initially, we posed two research questions:

[5]University policies govern the length of this period, and require a fee to be paid for late submissions.

- *RQ1*: What is the connection between a student's willingness to adhere to the schedule, the quality, and the functional correctness of the code they write?
- *RQ2*: Is identifying students facing difficulties early on possible?

We built a dataset from each student's activity during the semester to answer our research questions. We collected:

- Whether the student passed the home assignment or not;
- Dates of completion for onboarding, implementing, test design, and review phases;
- For onboarding commits: commit date;
- For implementing and review commits: commit date, performance on the hidden test suite, SonarQube quality metrics (SQALE index, numbers/types of bugs, code smells, written unit tests, and their coverage);
- For test design commits: commit date, test suite size, suite correctness (tests passing on the reference implementation/total tests), and suite completeness (mutants detected by suite/total mutants; incorrect tests excluded).

We utilized exploratory data analysis (EDA) methods to analyze patterns in the data, enabling us to examine correlations between student activity timelines, code quality metrics, and functional correctness. We also aimed to identify behavioral indicators of potential challenges during the assignment.

Initially, we excluded all student records involving plagiarism during the home assignment. This resulted in a dataset of 641 students for further analysis (refer to Figure 3). Among these, 69% completed the onboarding phase on time, 29% were late, and 2% gave up before completion. Conversely, just 29% completed the implementing phase on time, with 60% late and 11% dropping out by this point. A trend was observed where those on time in onboarding often fell behind in implementing, whereas few who started late managed to catch up. In the test design phase, 37% were on time, 49% late, and 14% gave up before reaching it. Those delayed in implementation usually continued the pattern into the test design phase, although a few students caught up with the deadlines here. Notably, by the

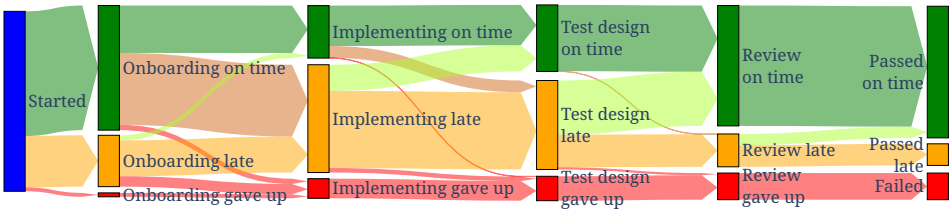
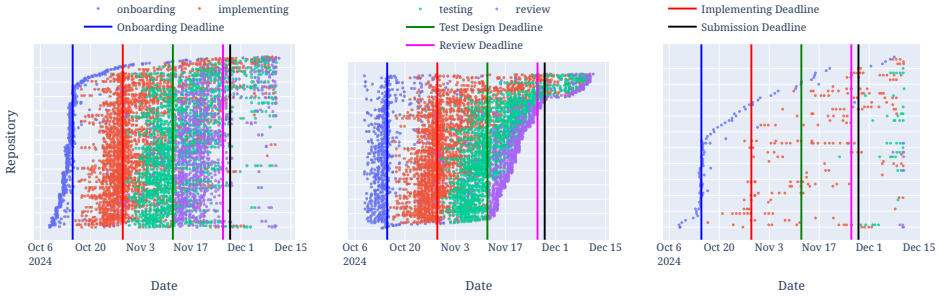


Figure 3. Sankey diagram visualizing the proportion of students completing the home assignment phases on time.

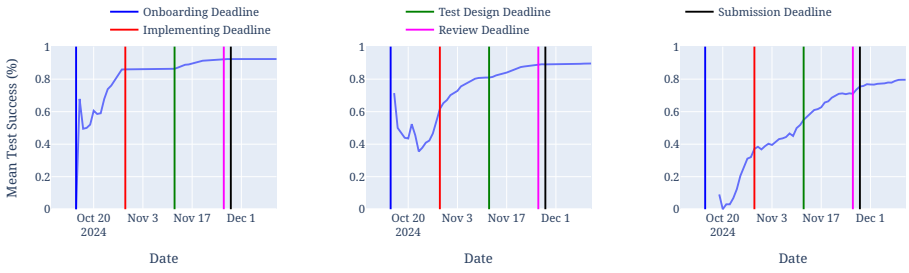
review deadline, 67% were on time, split evenly between those on time and those late in earlier phases, while 18% were late, and 15% gave up. In the two days from the optional review deadline to the final cutoff, an additional 6% completed their review phase and, consequently, the home assignment.

Investigating student progression in further depth (Figure 4), it becomes evident that although a single week sufficed for the onboarding phase, the implementation and test design phases demanded two weeks of student engagement. However, the majority initiated the implementation phase a mere week before its deadline (Figure 4a). Students who commenced the home assignment well ahead of the optional onboarding phase deadline largely managed to avoid late submissions (Figure 4a), whereas late submitters frequently did not commence the onboarding and implementation phases on schedule (Figure 4b). Additionally, Figure 4c shows that nearly all failing students only began the test design phase during the late submission timeframe.



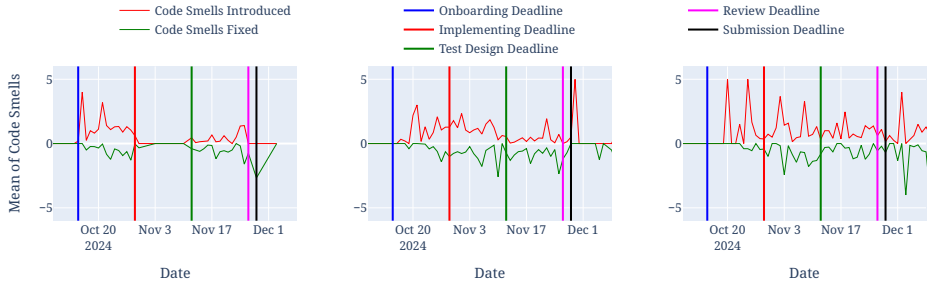
(a) ...in the order of the first commit. **(b)** ...in the order of the last commit. **(c)** ...who failed.

Figure 4. Scatterplot where each point represents a commit of a student's repository...



(a) ...by the implementation deadline. **(b)** ...by the test design deadline. **(c)** ...late.

Figure 5. The mean test success percentage on each day, faceted by the completion of the coding phase...



(a) ...by the implementation deadline.

(b) ...by the test design deadline.

(c) ...late.

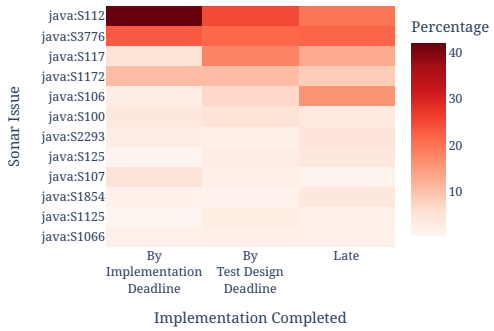
Figure 6. The mean number of code smells introduced (red, above zero) and fixed (green, below zero) on each day, faceted by the completion of the coding phase...

Regarding functional correctness, Figure 5 illustrates students' mean test success rate. Those who completed the implementation by its deadline (Figure 5a) attained an 86% success rate, ultimately rising to 93% after addressing issues revealed by their peers' test suites during the review phase. Conversely (Figure 5b), students who completed the implementation only by the test design deadline, being at most 2 weeks late, achieved 80% by that time and improved to 90% by the assignment's conclusion. Thus, students with delayed completion demonstrated worse functional correctness, requiring more revisions during review. Students who missed even the test design deadline (Figure 5c) reached only 80% by the assignment's end.

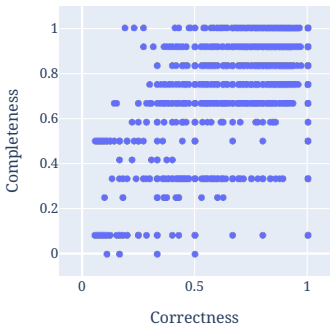
Figure 6 depicts the average change in code quality among students. It reveals that students who met the implementation deadline frequently generated issues during this phase, which they subsequently addressed during the review stage (see Figure 6a). Those completing the implementation by the test design deadline (delayed by up to two weeks) introduced issues over a prolonged period but actively resolved them (refer to Figure 6b) later. Students delayed by more than two weeks not only prolonged issue introduction but also created more issues on average and resolved fewer than the aforementioned groups (Figure 6c).

Analyzing the code quality further, Figure 7a illustrates the most common code smells observed at the end of the project. Table 1 reveals that the second most frequent issue was the high cognitive complexity of student code, indicating poor structural organization. The primary issue among punctual students involved throwing generic exceptions. Notably, students who started late threw fewer exceptions, suggesting a potential neglect in addressing edge cases. Further distinctions include delayed students' non-compliance with naming conventions and a tendency to retain debugging logs in final submissions.

An examination of individual outcomes corroborates prior results. Figure 8 compares the students' project code quality, assessed with SonarQube's SQALE



(a) The heatmap of the frequency of the most common code smells found in the students' code.



(b) The correctness of the students' designed test suite depicted against the completeness of the test suite

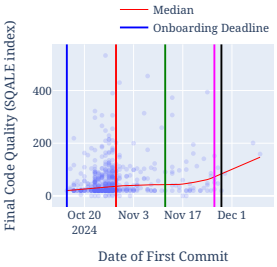
Figure 7

Table 1. The description, severity, and category of the most common code smells found in the student's code.

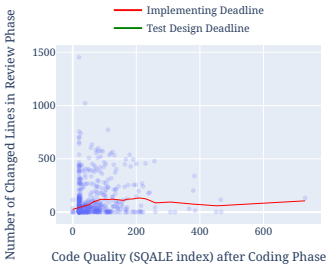
Rule	Description	Severity	Category
java:S112	Generic exceptions should never be thrown	Medium	Intentionality
java:S3776	Cognitive complexity too high	High	Adaptability
java:S117	Variable naming convention not complied with	Low	Consistency
java:S1172	Unused parameters	Medium	Intentionality
java:S106	Logging to standard output	Medium	Adaptability
java:S100	Method naming convention not complied with	Low	Consistency
java:S2293	Diamond operator not used	Low	Intentionality
java:S125	Commented out block of code	High	Intentionality
java:S107	Method has too many parameters	High	Adaptability
java:S1854	Unused assignments	High	Intentionality
java:S1125	Redundant boolean literals	Low	Consistency
java:S1066	Mergeable if statement	High	Intentionality

index, against other measures. Figure 8a illustrates a pattern where projects initiated later show poorer final code quality, particularly those starting near deadlines, implying students sacrificed quality for adherence to the timeline, and chose not to address code quality later. Figure 8b reveals that a higher SQALE index at the end of the implementation indicates a higher number of lines changed in the review phase, suggesting higher challenges in resolving code issues post-implementation. While the SQALE index goes from 0 to 200, the median line changes increase from 20 to 120. Insufficient amount of data exist for higher SQALE indices. Additionally, Figure 8c correlates lower code quality with reduced test success percentage, indicating a decline in functional correctness.

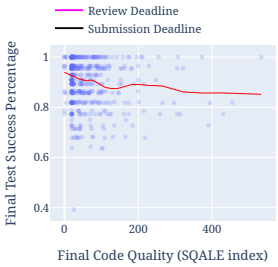
Finally, Figure 7b presents the two evaluation criteria for the student-designed test suites. Although there is a noticeable relationship where increased correctness suggests greater completeness and the reverse, the considerable variance indicates that students probably focused on one metric at a time.



(a) When did the students start the assignment, vs. what was their final code quality.



(b) What was the students' code quality at the end of the coding phase, vs. how much did they modify their implementation in the review phase.



(c) What was the students' final code quality compared to how much of the hidden test suite did their implementation pass at the end.

Figure 8. The connection between the students' work schedule, code quality, and functional correctness.

Discussion. Prior results demonstrate a clear connection between students' adherence to the schedule, code quality, and functional correctness (*RQ1*). A delayed start on the assignment often results in diminished code quality and increased review workloads, which adversely affects functional correctness. However, late starters may possess lower programming skills, contributing to these outcomes. Evaluations would benefit from incorporating data from previous programming courses to exclude this option. Conversely, no definitive early indicators of failure are evident early on (*RQ2*). Although failing students frequently delay their start, this is also true of several students who ultimately succeed, necessitating further investigation.

4. AI-assisted outcome estimation

To deepen our analysis of *RQ2*, whether students facing difficulties can be identified early, we applied an Explainable Boosting Machine (EBM) [14], a transparent, interpretable model that balances predictive performance with explainability. Unlike traditional decision tree-based approaches, EBM often achieves higher accuracy while allowing detailed insight into how individual features influence predictions. This makes it particularly valuable in educational settings, where understanding the reasoning behind model decisions is essential. By examining feature contributions and interaction effects, we aim to uncover which early behaviors most indicate eventual failure, enabling more informed, data-driven interventions.

Following data cleansing and preparation, we generated eight distinct datasets. Weekly thresholds aligning with the phase deadlines and midpoints were established (refer to Table 2), and all data beyond these thresholds were filtered out to finalize

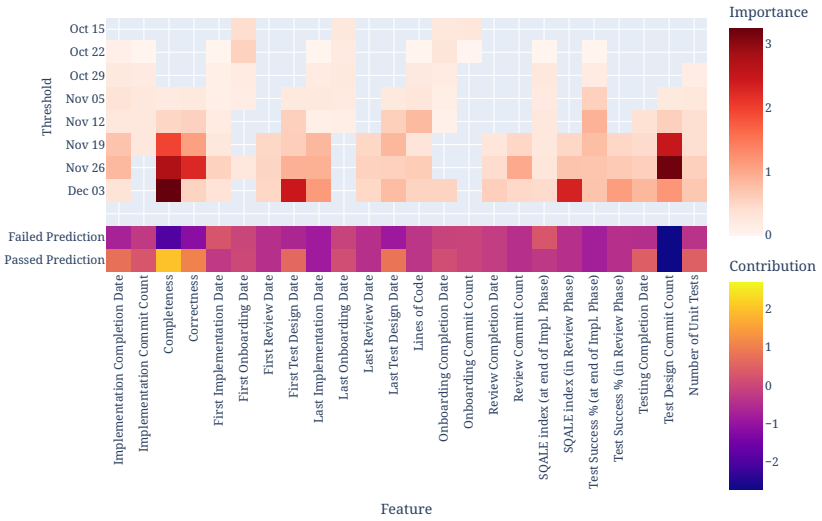


Figure 9. The importance of certain features of the explainable boosting classifier (Importance). The contribution of features to a failing and passing prediction for the sixth dataset (Contribution).

our datasets. Thus, each dataset contained everything known at that threshold. The target variable was whether the student passed the home assignment.

Then, an EBM-based binary classifier was trained for each dataset. Figure 9 presents the training outcomes, whereas Table 2 displays the AUC, sensitivity, and specificity metrics.

The specificity is around 0.5 for the initial two datasets aligned with the onboarding deadline and implementation phase midpoint. This indicates the model failed to identify reasons for student failure, consistently predicting a passing outcome. Referencing Figure 4, merely half of the failing students had commenced work by these points. Those engaged predominantly had not begun the implementation phase. Furthermore, given that numerous students with analogous behavior completed the home assignment successfully, the classifier lacked sufficient data for differentiation.

From the third dataset onward, specificity steadily rises with the addition of more information, while sensitivity slightly diminishes due to a substantial number of new student entries at this stage. For the third and fourth datasets, as the implementation phase concludes and the test design phase reaches its midpoint, specificity surpasses 70%. Key predictors for these datasets include the completion date of the implementation phase, functional correctness, the number of commits, and lines of code authored by students in this phase. Notably, from the fourth dataset, the number of commits in the test design phase and the accuracy and completeness of the test suite become increasingly significant.

Table 2. The AUC, sensitivity, and specificity of the models.

Threshold	AUC	Sens.	Spec.	Threshold	AUC	Sens.	Spec.
October 15	0.740	0.829	0.533	November 12	0.949	0.876	1.000
October 22	0.825	0.924	0.567	November 19	0.967	0.914	1.000
October 29	0.876	0.810	0.700	November 26	0.985	0.971	1.000
November 05	0.929	0.876	0.833	December 03	0.994	0.962	1.000

The fifth (test design deadline) is the first dataset with a specificity of 1.0, demonstrating the model's capability to predict failing students accurately. From this point, sensitivity and AUC rise and remain high, signifying strong predictive performance. Previous trends persist: crucial elements include the commit count during the test design phase and the test suite's correctness and completeness. Additionally, functional correctness, the implementation phase's line count, and the final test design phase commit date gain significance.

Datasets six and seven, corresponding to the review phase's midpoint and deadline, emphasize test suite correctness, completeness, and commit count during test design phase as critical. Notably, the implementation phase's completion date and review phase commit count become more significant than in other datasets. In the seventh dataset, implementation phase features, like lines of code, initial commit date, and SQALE index, become notable. This pattern, shown in Figure 4c, reflects many students who postponed implementation and failed the course.

By the eighth dataset related to late submission, over 75% of students had already completed the home assignment. Key characteristics align with the seventh dataset, but the final commit date in the test design phase, the final SQALE index, implementation phase code lines, and test success percentage (functional correctness) gain more significance.

Using an EBM facilitates the explanation of each feature's contribution to predictions. This is illustrated with a failing and a passing prediction for the sixth dataset (Figure 9, bottom two rows). Negative contributions are linked with failures, while positive ones signify successes. These contributions are largely consistent with the importances in the datasets. For students who fail, this insight aids in giving customized advice aimed at passing the assignment. Conversely, for successful students, it can be used to suggest improvements in features that contributed against that prediction, improving the quality of their solution.

Discussion. Our findings demonstrate the potential for identifying students who are encountering difficulties at an early stage (*RQ2*). However, there is a notable amount of false negative outcomes during the initial month of observation. As depicted in Figure 4, there is an implication that prioritization of support should be directed towards students who show a lack of engagement with their assignments within this initial timeframe. Nevertheless, EBMs continue to serve effectively in the context of prediction interpretation, providing valuable insights into the primary features that contribute to the outcomes. Analyzing these features, we can provide automated recommendations tailored for students who are at risk of failing.

5. Related work

Multiple tools provide interactive and gamified learning experiences tailored for different audiences and skill levels. LearnGitBranching [22] is an example that elucidates the fundamentals of distributed version control. Codio and CodeAcademy cater to professionals, students, and universities [5, 20], covering a wide range of topics. While these platforms offer a simulated environment for learning, they are typically restricted to specific domains, lacking real-world project complexities. CI-based solutions [1, 2, 6, 15], utilizing GitHub Actions, GitLab CI/CD, and Jenkins, are prevalent due to their capacity for continuous, automated feedback, urging students to use industry-standard tools. However, these systems can be limited by the CI platform, presenting challenges in database integration, meeting deadlines, and concealing evaluative aspects from students. Our method aims to integrate the interactive and gamified aspects of simulation tools with the technological stack of CI-based systems.

Extensive research exists on teaching code quality within software engineering education [9, 11]. Gilson et al. [7] examine software quality assessment in a year-long project, emphasizing long-term technical debt management by students. Conversely, this paper assesses code quality in relation to other metrics, such as functional correctness and deadline compliance, over a shorter 7-week timeframe. Meanwhile, Senger et al. [19] investigate the links between assignment solution time, final outcomes, and code quality in small tasks. A notable distinction is that Senger et al. do not provide static analysis results to students and do not aim to teach static analysis.

Most recent research investigating the use of static analysis tools in software engineering education, such as [12, 13, 18], focuses on the security vulnerability detection. In contrast, our paper exclusively focuses on maintainability (SQALE index) and functional correctness.

6. Conclusion

In summary, our assignment framework demonstrates the feasibility and effectiveness of incorporating real-world software engineering practices into educational settings at scale. Utilizing industry-standard tools, best practices, and automated feedback, we offer a learning experience that goes beyond traditional programming tasks, enhancing student participation and readiness for professional software engineering. Our evaluation indicates a correlation between students' willingness to meet optional deadlines and the quality and functionality of their code, and shows that AI tools can identify struggling students during the assignment period. In the future, we plan to use AI-assisted outcome predictions to provide tailored advice for students who are falling behind.

References

[1] X. Bai, M. Li, D. Pei, S. Li, D. Ye: *Continuous delivery of personalized assessment and feedback in agile software engineering projects*, in: Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, ICSE-SEET '18, New York, NY, USA: Association for Computing Machinery, 2018, pp. 58–67, ISBN: 9781450356602, DOI: 10.1145/3183377.3183387.

[2] J. Bennedsen, T. Böjttjer, D. Tola: *Using GitHub classroom in teaching programming*, in: Proceedings of the 18th International CDIO Conference, 2022, p. 690.

[3] S. Bhalerao, D. Puntambekar, M. Ingle: *Generalizing Agile software development life cycle*, International journal on computer science and engineering 1.3 (2009), pp. 222–226, ISSN: 0975–3397.

[4] J. C. Cortés Ríos, S. M. Embury, S. Eraslan: *A unifying framework for the systematic analysis of Git workflows*, Information and Software Technology 145 (2022), p. 106811, ISSN: 0950-5849, DOI: 10.1016/j.infsof.2021.106811.

[5] D. Croft, M. England: *Computing with Codio at Coventry University: Online virtual Linux boxes and automated formative feedback*, in: Proceedings of the 3rd Conference on Computing Education Practice, CEP '19, New York, NY, USA: Association for Computing Machinery, 2019, ISBN: 9781450366311, DOI: 10.1145/3294016.3294018.

[6] B. P. Eddy, N. Wilde, N. A. Cooper, B. Mishra, V. S. Gamboa, K. M. Shah, A. M. Deleon, N. A. Shields: *A Pilot Study on Introducing Continuous Integration and Delivery into Undergraduate Software Engineering Courses*, in: 2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T), 2017, pp. 47–56, DOI: 10.1109/CSEET.2017.18.

[7] F. Gilson, M. Morales-Trujillo, M. Mathews: *How junior developers deal with their technical debt?*, in: Proceedings of the 3rd International Conference on Technical Debt, TechDebt '20, Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 51–61, ISBN: 9781450379601, DOI: 10.1145/3387906.3388624.

[8] Y. Jia, M. Harman: *An Analysis and Survey of the Development of Mutation Testing*, IEEE Transactions on Software Engineering 37.5 (2011), pp. 649–678, DOI: 10.1109/TSE.2010.62.

[9] H. Keuning, J. Jeuring, B. Heeren: *A Systematic Mapping Study of Code Quality in Education*, in: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2023, Turku, Finland: Association for Computing Machinery, 2023, pp. 5–11, ISBN: 9798400701382, DOI: 10.1145/3587102.3588777.

[10] B. McHugh: *Beyond transparency: open data and the future of civic innovation*, in: Code for America Press San Francisco, 2013, chap. Pioneering open data standards: The GTFS Story, pp. 125–135, ISBN: 978-0615889085.

[11] D. Nikolić, D. Stefanović, M. Nikolić, D. Dakić, M. Stefanović, S. Koprivica: *Uncovering Determinants of Code Quality in Education via Static Code Analysis*, IEEE Access 12 (2024), pp. 168229–168244, DOI: 10.1109/ACCESS.2024.3426299.

[12] S. Nocera, S. Romano, R. Francese, G. Scanniello: *Software engineering education: Results from a training intervention based on SonarCloud when developing web apps*, Journal of Systems and Software 222 (2025), p. 112308, ISSN: 0164-1212, DOI: 10.1016/j.jss.2024.112308, URL: https://www.sciencedirect.com/science/article/pii/S0164121224003522.

[13] S. Nocera, S. Romano, R. Francese, G. Scanniello: *Training for Security: Results from Using a Static Analysis Tool in the Development Pipeline of Web Apps*, in: Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training, ICSE-SEET '24, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 253–263, ISBN: 9798400704987, DOI: 10.1145/3639474.3640073.

[14] H. Nori, S. Jenkins, P. Koch, R. Caruana: *InterpretML: A Unified Framework for Machine Learning Interpretability*, ArXiv abs/1909.09223 (2019), URL: https://api.semanticscholar.org/CorpusID:202712518.

[15] M. Ohtsuki, K. Ohta, T. Kakeshita: *Software engineer education support system ALECSS utilizing DevOps tools*, in: Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services, iiWAS '16, New York, NY, USA: Association for Computing Machinery, 2016, pp. 209–213, ISBN: 9781450348072, DOI: 10.1145/3011141.3011200.

[16] S. Ouhbi, N. Pombo: *Software Engineering Education: Challenges and Perspectives*, in: 2020 IEEE Global Engineering Education Conference (EDUCON), 2020, pp. 202–209, DOI: 10.1109/EDUCON45650.2020.9125353.

[17] N. B. Ruparelia: *Software development lifecycle models*, ACM SIGSOFT Software Engineering Notes 35.3 (2010), pp. 8–13, DOI: 10.1145/1764810.1764814.

[18] A. Sanders, G. S. Walia, A. Allen: *Assessing common software vulnerabilities in undergraduate computer science assignments*, Journal of The Colloquium for Information Systems Security Education 11.1 (2024), pp. 8–8, DOI: 10.53735/cisse.v11i1.179.

[19] A. Senger, S. H. Edwards, M. Ellis: *Helping Student Programmers Through Industrial-Strength Static Analysis: A Replication Study*, in: Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1, SIGCSE 2022, Providence, RI, USA: Association for Computing Machinery, 2022, pp. 8–14, ISBN: 9781450390705, DOI: 10.1145/3478431.3499310.

[20] J. H. Sharp: *Using Codecademy Interactive Lessons as an Instructional Supplement in a Python Programming Course.* Information Systems Education Journal 17.3 (2019), pp. 20–28.

[21] I. Sommerville: *Software Engineering*, International Computer Science Series, Pearson, 2011, ISBN: 9780137053469.

[22] G. Wagner, L. Thurner: *Teaching Tip: Rethinking How We Teach Git: Pedagogical Recommendations and Practical Strategies for the Information Systems Curriculum*, Journal of Information Systems Education 36.1 (2025), pp. 1–12, DOI: 10.62273/BTKM5634.

[23] T. Winters, T. Manshreck, H. Wright: *Software engineering at Google: Lessons learned from programming over time*, O'Reilly Media, Inc., 2020.